Tor
Extensible Search with Hookable Disjunction

Tom Schrijvers
Ghent University
tom.schrijvers@ugent.be

Markus Triska
Vienna University of Technology
triska@dbai.tuwien.ac.at

Bart Demoen
KU Leuven
bart.demoen@cs.kuleuven.be

Abstract

Horn Clause Programs have a natural depth-first procedural semantics. However, for many programs this procedural semantics is ineffective. In order to compute useful solutions, one needs the ability to modify the search method that explores the alternative execution branches.

Tor, a well-defined hook into Prolog disjunction, provides this ability. It is light-weight thanks to its library approach and efficient because it is based on program transformation. Tor is general enough to mimic search-modifying predicates like ECLiPSe's `search/6`. Moreover, Tor supports modular composition of search methods and other hooks. Our library is already provided and used as an add-on to SWI-Prolog.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages

Keywords Prolog, disjunction, search, modularity

1. Introduction

Kowalski's well-known adage [9] crisply captures the essence of programming in the equation:

ALGORITHM = LOGIC + CONTROL

In Prolog, the LOGIC part is captured in the programmer-supplied rules or clauses that have a first-order logic interpretation. The CONTROL component is supplied by the Prolog engine and essentially consists of *search*. In order to answer queries, a Prolog engine performs a backward-chaining depth-first tree search.

Prolog's default search strategy is in practice inadequate to effectively scour large search spaces. As a consequence, the programmer often has to complement Prolog's CONTROL with additional hints or heuristics in the form of extra code. This is particularly prevalent in the context of Constraint Logic Programming where it is common practice for the programmer to complement a constraint model with a search specification.

Unfortunately, it is not all that easy to cleanly separate LOGIC and CONTROL when implementing search heuristics in Prolog. When one discovers that Prolog's CONTROL is ineffective, it is often impossible to orthogonally add one's own CONTROL without touching the existing LOGIC. The problem is that syntactically logic and control in Prolog are generally tightly coupled, and adding a different control means cross-cutting existing code.

In this paper we present a novel approach to adding, in an orthogonal manner, CONTROL. Our solution features the following properties:

- It is a light-weight library-based approach that is easily portable to different Prolog systems: it is currently an SWI-Prolog library [22]
- Our approach has all the benefits of modularity: search methods can be composed and the library of these heuristics is (user-)extensible.
- Its overhead is minimal, as we demonstrate on benchmarks: this is achieved through `term_expansion/2`, a feature present in most Prolog systems.

With Tor, we capture all common search methods in CLP(FD) libraries such as ECLiPSe's `search/6` [13]. This approach is indeed particularly suitable for Constraint Logic Programming, but also useful for general Prolog programs with a large search space.

2. Problem Statement

We illustrate the heart of the matter on a simple labeling predicate `label/1` written against SWI-Prolog's `clpfd` library [18] (see Fig. 1, left). `label/1` defines a search tree where the branches are created by the disjunction.[1]

Suppose that for a certain call `label([`X_1`,...,`X_n`])` the search tree is too large to fully explore. In order to get some useful answers, we may decide to leave certain parts of the tree unexplored, conceptually pruning the tree. One particular way in which we can do this is by reaping the low-hanging solutions only, and pruning the subtrees that are below a certain depth. Hence, we impose a *depth bound* on Prolog's depth first search. This is achieved by modifying the `label/1` code to that of `label/2` (Fig. 1, right) where the second parameter is the depth bound.

Imposing a depth bound may or may not be a successful approach to getting useful answers. If it turns out to be unsuccessful, we can try other pruning strategies like imposing a node bound or a discrepancy bound. Each of these requires rewriting the `label/1` predicate to incorporate a different pruning technique. In general, we follow an explorative process whereby we write and evaluate several different variants of the labeling code until we find an effective pruning strategy.

PPDP'12, September 19–21, 2012, Leuven, Belgium.
Copyright © 2012 ACM 978-1-4503-1522-7/12/09...$5.00.

[1] `fd_inf/2` returns the smallest value in a variable's finite domain.

```
label([]).
label([Var|Vars]) :-
  ( var(Var) ->
      fd_inf(Var,Value),
      ( Var #= Value,
        label(Vars)
      ;
        Var #\= Value,
        label([Var|Vars])
      )
  ;
      label(Vars)
  ).
```

```
label([] ,_ ).
label([Var|Vars] ,D ) :-
  ( var(Var) ->
      D > 0,
      ND is D - 1,
      fd_inf(Var,Value),
      ( Var #= Value,
        label(Vars ,ND )
      ;
        Var #\= Value,
        label([Var|Vars] ,ND )
      )
  ;
      label(Vars ,D )
  ).
```

Figure 1. Labeling predicate: plain (left) and with depth bound (right).

2.1 Problems with this Approach

The problems with the above approach should be apparent:

- The approach follows the well-known copy-paste-modify anti-pattern. Variants of the labeling code are copied all over the place, potentially propagating bugs and rendering maintenance into a nightmare. Working code is modified.
- The same heuristic is implemented over and over in different settings (different applications, different labeling predicates, different Prolog systems, ...). This process is error-prone, wastes precious programmer time and is bound to yield non-optimal code quality.
- The effort and expertise required to combine working labeling code with various search heuristics is non-trivial. This means that fewer combinations are explored by programmers under time pressure or unfamiliar with particular heuristics. The end result is that suboptimal solutions are obtained.
- As soon as the labeling code spans several different predicates or multiple invocations of the same predicate, the complexity of adding search heuristics increases drastically.

2.2 Current Solutions

Most of the current solutions are specific to CLP, but we are aware of one general Prolog approach.

CLP Solutions In the context of CLP **ECLiPSe** [13] copes with this problem by providing a number of search methods in the `search/6` predicate. This predicate lets the user control through its various arguments the selection method, the choice method and the search method: the former two decide on which variable is used during labeling, and which value it is assigned first. They do not concern us here. The search method controls how the search tree is explored, e.g., depth-bounded, node-bounded or limited discrepancy search. Apart from individual search methods, only a fixed number of compositions is supported, such as changing strategy when a depth bound is reached. In this setting users can extend the set of supported heuristics and combinations by reprogramming parts of the `search/6` predicate.

We see the same approach in other Prolog systems' CLP(FD) libraries, albeit to a more limited extent. **SICStus** Prolog [2] allows imposing discrepancy and time limits, and **B-Prolog** [23] provides only a time limit. **GNU Prolog** [3] and **Ciao**'s new `clpfd` library provide no limits on top of depth-first search.

All CLP(FD) libraries do provide one extra search method: optimization with respect to an objective value. Optimization is typically implemented as either branch-and-bound or by restarting the whole search with a new bound whenever a solution is found.

Typically these approaches only support adding search heuristics to a simple goal made up of a labeling predicate defined in the corresponding CLP library. This means that complex goals made up of a conjunction of labeling calls or custom labeling predicates are not supported.

Prolog Solution We are aware of one other approach to modify Prolog's own search method: the breadth-first and iterative deepening program transformations in Ciao [6]. These modify annotated predicates in place and are not compositional.

All in all the available library support that Prolog systems provide is very limited indeed. As soon as users face a constraint problem that requires a non-trivial search method, they are forced to write all their search code from scratch, and it can be very daunting to combine different search methods.

2.3 Our Solution

We propose to solve the above modularity problem concerning search methods by decoupling the code that defines the *search tree* (e.g., `labeling/1`) from the code that defines the *search method* (e.g., `depth_bound/2` and `lds/1`[2]). We emphasize that solver-specific heuristics for defining the search tree (e.g., variable and value selection strategies for CLP(FD)) are not in the scope of Tor. Tor's search methods specify how a search tree is visited, e.g., what parts are pruned.

Tor combines the search tree and search method specifications by means of the `search/1` predicate. For instance, we express three different scenarios as:

```
?- search(label([X1,...,Xn])).
?- search(depth_bound(10,label([X1,...,Xn]))).
?- search(lds(label([X1,...,Xn]))).
```

This approach does not suffer from the many disadvantages of the copy-paste-modify approach discussed above. In particular, the code for `label/1` is not touched, and the code for `depth_bound/2`

[2] `lds` stands for `limited-discrepancy search`

and `lds/1` is provided in a library or supplied by the user. Existing search tree descriptions and search methods are easily and independently reused and maintained, with all the benefits of increased productivity and code quality.

Moreover, our approach is truly compositional. With the same three components we can express an additional search heuristic that combines a depth bound with lds:

```
?- search(depth_bound(10,lds(label([X1,...,Xn])))).
```

At the same time, it is not limited to a single invocation of `label/1`. For instance, we can apply a search heuristic to a conjunction of labelings.

```
?- search(depth_bound(10,lds((label([X1,...,Xn])
                             ,label([Y1,...,Ym]))))).
```

Only small syntactic changes are necessary to use our highly modular library based approach: Instead of using Prolog's disjunction `(;)/2`, the search tree description must use our library's disjunction predicate `tor/2`.

In the next section we reveal the technical details of our solution.

3. Basic Solution

Many variants of Prolog's default search method are most easily expressed as an action to be taken at the moment the alternative branches are entered, and possibly modifying the branches themselves.

Basically, we achieve this in a general way by providing a hook into disjunction as follows:

- programmers load our `library(tor)` and use the Tor-disjunction at appropriate places in their code, i.e., instead of `(;)/2` they use `tor/2`
- the library contains a definition of the Tor-disjunction – this definition can be partially evaluated away in many cases
- the library provides a means to specify what action needs to be taken in entering one of the tor-disjunctive branches: the definition of the Tor-disjunction uses these actions
- the system provides a set of useful and common actions; the user can implement additional actions himself, without the need to modify the library; neither does the user need to know to which program this new action will be applied

We illustrate the basics of Tor on the labeling example with depth bound.

```
tor_label([]).
tor_label([Var|Vars]) :-
  ( var(Var) ->
      fd_inf(Var,Value),
      (    Var #= Value,
           tor_label(Vars)
      tor
           Var #\= Value,
           tor_label([Var|Vars])
      )
  ;
      tor_label(Vars)
  ).
```

Instead of the regular Prolog disjunction we use Tor's `tor/2` disjunction for non-determinism. This predicate is defined as:

```
G1 tor G2 :-
  ( b_getval(left,Left),
    call(Left,G1)   % conceptually: Left(G1)
  ;
    b_getval(right,Right),
    call(Right,G2)  % conceptually: Right(G2)
  ).
```

This definition provides two hooks into the disjunction by means of global variables `left` and `right`.[3] In these hooks the programmer installs *handlers* for the left and right branches to control the search. These handlers are *higher-order* predicates that take a goal and execute it in a (possibly) modified manner.

We obtain standard Prolog disjunction, if we use `call/1` as handler.

```
?- findall(X, ( X in 1..10
              , b_setval(left,call)
              , b_setval(right,call)
              , label([X])
              ), Values).
Values = [1,2,3,4,5,6,7,8,9,10].
```

Things get more interesting when we use a different handler. For instance, the following handler limits the depth:

```
dbs_handler(Branch) :-
   b_getval(depth_limit,Depth),
   NewDepth is Depth - 1,
   NewDepth > 0,
   b_setval(depth_limit,NewDepth),
   call(Branch).
```

In addition to calling the `Branch` goal, this handler also maintains a depth limit in the global variable `depth_limit`. When this limit reaches zero, the handler *replaces* the branch goal by failure. In other words, the handler prunes the search tree below the depth limit.

We use this depth-bounded search handler as follows:

```
?- findall(X, ( X in 1..10
              , b_setval(depth_limit, 4)
              , b_setval(left,  dbs_handler)
              , b_setval(right, dbs_handler)
              , label([X])
              ), Values).
Values = [1,2,3,4].
```

The above way to specify and invoke handlers is clearly too clumsy. The next sections remedy this by introducing much more convenient infrastructure.

4. Advanced Infrastructure

The basic approach we have presented in the previous section is rather primitive to use directly. In this section, we provide a layer of additional infrastructure that makes using Tor a much more pleasant and high-level experience.

4.1 Implicit Disjunctions

The `tor/1` declaration implicitly adds Tor-disjunctions between the clauses of a predicate. For instance, the Tor variant of `member/2` can be written conventionally as:

```
:- tor tor_member/2.
tor_member(X,[X|_]).
tor_member(X,[_|Xs]) :- tor_member(X,Xs).
```

[3] Note that `b_getval/2` and `b_putval/2` are SWI-Prolog builtins for reading and writing global mutable variables, whose names are atoms. Their non-backtrackable counterparts are `nb_getval/2` and `nb_putval/2`.

instead of

```
tor_member(X,[Y|Ys]) :-
  (
      X = Y
  tor
      tor_member(X,Ys)
  ).
```

4.2 Default Handler

The convenient predicate `search/1` sets up the default handler for both hooks: `call/1`.

```
search(Goal) :-
  b_setval(left,call),
  b_setval(right,call),
  call(Goal).
```

With this default handler, `tor/2` corresponds simply to plain disjunction `(;)/2`.[4] For instance, with `search/1` we recover the behavior of `label/1` of Fig. 1 from the Tor-variant.

`search(tor_label(Vars))` $\equiv$ `label(Vars)`

4.3 Custom Handlers

In order to facilitate installing new handlers, we provide two convenient predicates.

Firstly, `tor_handlers/3` composes the currently installed handlers with the ones provided. Then it runs the provided goal and finally, it resets the installed handlers.

```
tor_handlers(Goal,Left,Right) :-
  b_getval(left,LeftHandler),
  b_getval(right,RightHandler),
  b_setval(left,compose(LeftHandler,Left)),
  b_setval(right,compose(RightHandler,Right)),
    call(Goal),
  b_setval(left,LeftHandler),
  b_setval(right,RightHandler).

compose(G1,G2,Goal) :- call(G1,call(G2,Goal)).
  % conceptually: G1(G2(Goal))
```

Section 5.8 shows that this approach enables composing different search methods.

Secondly, in many cases, the handler only needs to precede the actual branch goal by its own goal, much like *before-advice* in Aspect Oriented Programming [8]. For that purpose we introduce the `tor_before_handlers/3` predicate.

```
tor_before_handlers(Goal,Left,Right) :-
  tor_handlers(Goal,before(Left),before(Right)).

before(G1,G2) :- G1, G2.
```

Note that `before/2` is an alias for Prolog conjunction `(,)/2` which is convenient to use in partially applied form.

We illustrate the use of `tor_before_handlers/3` on the query example we've seen earlier:

```
?- findall(X, ( X in 1..10
              , b_setval(depth_limit, 4)
              , tor_before_handlers(label([X])
                                   ,dbs_handler
                                   ,dbs_handler)
              ), Values).
   Values = [1,2,3,4].
```

[4] Apart from the scope of any cuts in the alternative branches

where `dbs_handler/0` now is no longer responsible for calling the branch goal:

```
dbs_handler :-
   b_getval(depth_limit,Depth),
   NewDepth is Depth - 1,
   NewDepth > 0,
   b_setval(depth_limit,NewDepth).
```

We will see examples of `tor_handlers/3` in Section 5.

4.4 Reference Cells

Usually a handler maintains some *stateful information*. For instance, the depth bounded handler maintains the current depth in the search tree. Generally, it is considered good style to declaratively thread such state through predicate calls by means of input and/or output arguments. This is the case in the `label/2` predicate, where the second argument captures the current depth.

However, Tor decouples the handler code from the labeling code. This means there is no explicit static control flow from one invocation of the handler to the next. Hence, there is no way in which state arguments can be threaded explicitly. In order to overcome this problem, we turn to a non-declarative solution: the use of mutable variables. These mutable variables provide an implicit channel of communication between successive invocations of a handler.

While mutable variables are notoriously error-prone and detrimental to the overall declarative nature of a Prolog program, Tor aims to contain the danger to a reasonable level. In particular, users of off-the-shelf handlers should not be exposed directly or indirectly to the fact that mutable variables are used internally. Only handler implementors, i.e., advanced users of the Tor library, should deal with them.

Moreover, Tor provides handler implementors with *reference cells* for mutable variables. These prevent implementors from shooting themselves in the foot in the most obvious ways and handler instances from interfering in unexpected ways. To illustrate the problem, consider again the naive use of the mutable global variable `depth_limit`. This global variable is problematic for two important reasons:

- We cannot ensure that no other part of the program, e.g., a different handler, uses the same name for other purposes.
- We cannot use two instances of the same variable simultaneously, e.g., to impose one depth bound on the overall search tree and another one on the top part of the tree.

In both cases, the issue is one of interference. While we can consider the first case as somewhat unlucky, the latter clearly limits usability.

In order to avoid the interference problem we advocate the use of reference cells, implemented by means of mutable terms, as proposed by Aggoun and Beldiceanu [1]. Our implementation for creating, reading and writing such variables is as follows:

```
new_bvar(InitialValue,Var) :-
  var(Var),
  Var = bvar(InitialValue).

b_put(Var,Value) :-
  Var = bvar(_),
  setarg(1,Var,Value).

b_get(bvar(Value),Value).
```

The depth-bounded search method now becomes:

```
dbs(Depth,Goal) :-
  new_bvar(Depth,Var),
  tor_before_handlers(Goal,dbs_handler(Var)
                          ,dbs_handler(Var)).

dbs_handler(Var) :-
  b_get(Var,N),
  N > 0,
  N1 is N - 1,
  b_put(Var,N1).
```

Note that `dbs_handler/1` now takes the mutable variable as a parameter. We make use of a *wrapper* predicate `dbs/2` to create the mutable variable and install the handlers. Now our query example becomes very compact indeed:

```
  ?- findall(X, ( X in 1..10
               , dbs(4,label([X]))
               ), Values).
  Values = [1,2,3,4].
```

Because some handler information has to persist across backtracking, we also provide a non-backtrackable variant of reference cells.

```
new_nbvar(InitialValue,Var) :-
  var(Var),
  Var = nbvar(InitialValue).

nb_put(Var,Value) :-
  Var = nbvar(_),
  nb_setarg(1,Var,Value).

nb_get(nbvar(Value),Value).
```

We will see examples of their use in the next section.

5. Handler Library

With the Tor infrastructure, it is easy to write various search methods in a modular way. While the user can write custom ones himself, Tor already provides a substantial library of handlers. We cover several of them here.

5.1 Depth Bounded Search

We have already developed a depth-bounded search handler in the previous section. Here we only remind the reader that it recovers the behavior of `label/2` of Fig. 1 as follows:

```
search(dbs(Depth,tor_label(Vars)))
              ≡
       label(Vars,Depth)
```

5.2 Discrepancy-Bounded Search

The discrepancy-bounded search heuristic is a small variant of depth-bounded search: the bound is only updated in right branches.

```
dibs(Discrepancies,Goal) :-
  (dib,Var),
  new_bvar(Discrepancies,Var),
  tor_before_handlers(Goal,true,dbs_handler(Var)).
```

5.3 Iterative Deepening

Iterative deepening emulates breadth-first search by means of increasing depth-bounds. The implementation below makes use of two variables: `DVar` to keep track of the current depth, and `PVar` to record whether the depth limit has been enforced in the current iteration. The handler `id_handler/2` checks and updates these variables in every node of the search tree. The driver `id_loop/2` initiates each iteration and, if pruning occurred, starts the next one.

```
id(Goal) :-
  new_bvar(0,DVar),
  new_nbvar(not_pruned,PVar),
  id_loop(Goal,DVar,0,PVar).

id_loop(Goal,DVar,Depth,PVar) :-
  b_put(DVar,Depth),
  nb_put(PVar,not_pruned),
  Handler = id_handler(DVar,PVar),
  ( tor_before_handlers(Goal,Handler,Handler)
  ;
    nb_get(PVar,Value),
    Value == pruned,
    NDepth is Depth + 1,
    id_loop(Goal,DVar,NDepth,PVar)
  ).

id_handler(DVar,PVar) :-
  b_get(DVar,N),
  ( N > 0 ->
      N1 is N - 1,
      b_put(DVar,N1)
  ;
      nb_put(PVar,pruned),
      false
  ).
```

5.4 Limited Discrepancy Search and Factored Iteration

The traditional limited discrepancy search [5] is a minor variant of iterative deepening. It applies the depth-bound only in right branches. Put differently, limited discrepancy search is to discrepancy-bounded search what iterative deepening is to depth-bounded search.

With some abstraction, we can factor out the common iteration part of iterative deepening and limited discrepancy search:

```
iterate(PGoal) :-
  with_pruned(
    iterate_loop(0,PGoal)).

iterate_loop(N,PGoal) :-
  (
    call(PGoal,N)
  ;
    is_pruned,
    reset_pruned,
    M is N + 1,
    iterate_loop(M,PGoal)
  ).
```

This iteration pattern runs a goal `PGoal` that is parameterized in a natural number `N`. The goal uses this number as a bound and applies pruning when the bound is exceeded. The iteration repeatedly restarts the goal with successive values for `N` until the goal completes without pruning.

With this iteration pattern we can express iterative deepening and limited discrepancy search as follows:

```
id(Goal)  :- iterate(flip(dbs,Goal)).
lds(Goal) :- iterate(flip(dibs,Goal)).

flip(Goal,Y,X) :- call(Goal,X,Y).
```

There is only one complicating factor: we need to communicate the pruning from the handler to the iteration. Despite the negative aspects of global variables, we opt for them due to their minimally intrusive nature.

```
prune :-
  set_pruned(true),
  fail.

reset_pruned :-
  set_pruned(false).

is_pruned :-
  get_pruned(true).

get_pruned(Flag) :-
  nb_getval(pruned,Flag).

set_pruned(Flag) :-
  nb_setval(pruned,Flag).

with_pruned(Goal) :-
  get_pruned(OldFlag),
  ( reset_pruned,
    call(Goal)
  ;
    set_pruned(OldFlag),
    fail
  ).

pruned_union(true,_,true).
pruned_union(false,true,true).
pruned_union(false,false,false).
```

With the imperative ugliness hidden in the above definitions, `prune` elegantly replaces `fail` in the handler code:

```
dbs_handler(Var) :-
  b_get(Var,N),
  ( N > 0 ->
      N1 is N - 1,
      b_put(Var,N1)
  ;
      prune
  ).
```

5.5 Branch-and-Bound Optimization

This well-known optimization approach posts constraints in the intermediate nodes of the search tree to find increasingly better solutions. Our implementation uses Tor to access those intermediate nodes and generate increasingly larger values of the `Objective` variable. It uses two variables, `BestVar` and `CurrentVar`. The former keeps track of the overall best solution so far, while the latter is the solution that the current node tries to improve upon.

Both the overall and current best solution are initialized to a value smaller than the infimum of the objective variable's domain. Whenever a solution is found, the overall best solution is updated. Whenever we backtrack into a Tor choicepoint, the handler synchronizes the current best solution with the overall best solution. If the current best solution was out of sync, the handler also imposes a new lower bound on the objective variable. Note that `inf` denotes negative infinity.

```
bab(Objective,Goal) :-
  fd_inf(Objective,Inf),
  Best is Inf - 1,
  new_nbvar(Best,BestVar),
  new_bvar(Best,CurrentVar),
  Handler = bab_handler(Objective,BestVar,CurrentVar),
  tor_before_handlers(Goal,Handler,Handler),
  nb_put(BestVar,Objective).

bab_handler(Objective,BestVar,CurrentVar) :-
  nb_get(BestVar,Best),
  b_get(CurrentVar,Current),
  ( Best \= inf , (Current == inf ; Best > Current ) ->
      Objective #> Best,
      b_put(CurrentVar,Best)
  ;
      true
  ).
```

5.6 Search Tree Observation

Tor does not only allow us to manipulate the traversal of the search tree in various search heuristics. It also enables us to observe the search tree in different ways in order to gain insight in the search process for (performance) debugging purposes.

Statistics Similar to SWI-Prolog's `profile/1`, `time/1` and `statistics/0` predicates, we can provide different components that monitor various metrics of the search tree and provide us with a convenient summary.

```
?- length(Xs,4), Xs ins 1..4,
   search(statistics((tor_label(Xs),writeln(Xs)))),
   false.
...
[4,4,4,4]
% Number of solutions: ....... 256
% Number of nodes: ........... 510
% Number of failures: .......... 0
```

The code for `statistics/1` is in the Tor library.

To support users who want to check whether they have successfully replaced all regular disjunctions with Tor, we also provide a tool that uses SWI-Prolog's choice point inspection primitives (like `prolog_current_choice/1`) to verify this.

Visualisation In addition to summarized data of the search tree, we can also visualize the actual search tree itself. For that purpose, we provide a predicate that emits a textual representation, a log, of the search tree:

```
log(Goal) :-
  tor_handlers(Goal,log_handler(left)
                   ,log_handler(right)),
  writeln(solution).

log_handler(Side,Goal) :-
  ( writeln(Side),  call(Goal)
  ;
    writeln(false), false).
```

A complimentary tool that turns this log into a PDF image is also available from our public code repository.

Fig. 2 shows the complete search tree for labeling 3 variables with domains of size 3 that are not involved in any constraints. The symbol $\top$ denotes that a solution is found at this node.

Fig. 3 shows two search trees for the 8-queens puzzle: The left one was created with depth limit (search strategy `dbs`) 4, and the right one with depth limit 7, where we stopped the search after finding the first solution. The symbol $\perp$ denotes pruning due to constraint propagation, and ! denotes a node that is not explored because the depth limit is exceeded at this level of the search tree.

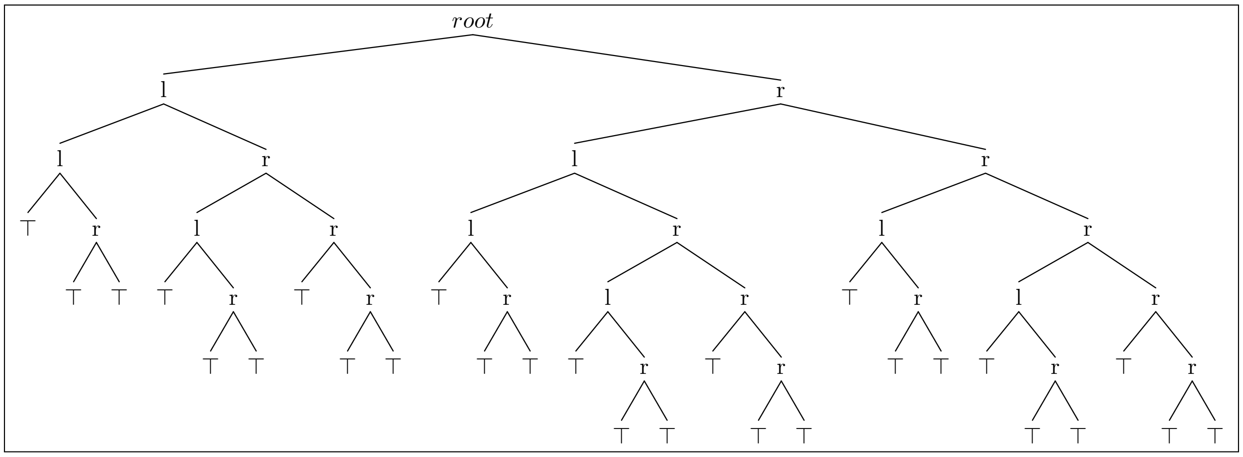
Figure 2. Search tree of `Xs = [_,_,_], Xs ins 1..3, search(log(tor_label(Xs)))`

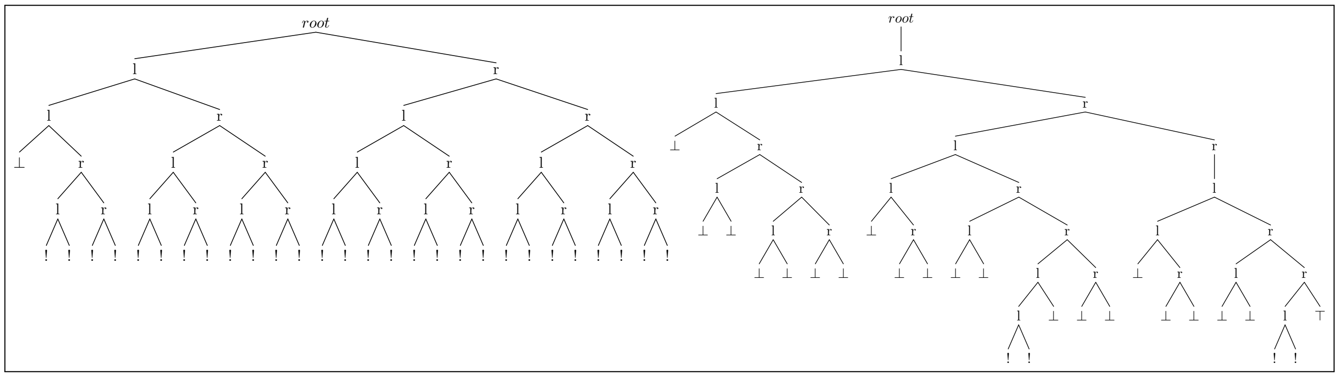
Figure 3. Search trees of 8-queens with depth bound 4 and 7

5.7 More and Higher-Order Handlers

We have implemented many other orthogonal search methods with tor, including all those offered by ECLiPSe's `search/6` predicate. In particular, several *higher-order search methods* are provided; these are search methods that are parametrized by other search methods.

An example of that is the following `dbs/3` variant on depth-bounded search. When it reaches the depth bound, it does not prune the remaining subtree, but activates the search method `Method`. A typical example is to limit the discrepancy once we reach a certain level in the search tree. This is achieved with `dbs(Level,lds(Discrepancies),Goal)`.

```
dbs(Level, Method, Goal) :-
  new_bvar(yes(Level),Var),
  tor_handlers(Goal,dbs_handler(Var,Method)
                  ,dbs_handler(Var,Method)).

dbs_handler(Var,Method,Goal) :-
  b_get(Var,MDepth),
  dbs_handler_(MDepth,Var,Method,Goal).

dbs_handler_(yes(Depth),Var,Method,Goal) :-
  ( Depth > 1 ->
      NDepth is Depth - 1,
      b_put(Var,yes(NDepth)),
      call(Goal)
  ;
      b_put(Var,no),
      call(Method,Goal)
  ).
dbs_handler_(no,_,_,Goal) :-
      call(Goal).
```

We recover the original first-order search method `dbs/2` with `dbs(Level,prune,Goal)` where:

```
prune(Goal) :- prune.
```

In ECLiPSe, only a fixed number of parameters can be supplied to these higher-order search methods, and `search/6` explicitly caters for each separate combination in its implementation. Not so with Tor. There is no restriction on the possible combinations; the higher-order search methods are truly parametric.

See our online code supplement for more handlers: `http://users.ugent.be/~tschrijv/tor`.

5.8 Composing Handlers

The beauty of Tor is that handlers can be composed. Below, we illustrate three kinds of composition.

Firstly, several handlers can be active at the same Tor-disjunction. For instance, we can perform branch-and-bound optimization with a depth-bound as follows:

```
?- ...,
   search(bab(Objective
             ,dbs(DepthLimit,tor_label(Vars)))).
```

Secondly, different handlers can be used for different parts of the search space. For instance, we can label the Xs and Ys variables each with their own depth limit:

```
?- ...,
   search((dbs(XsLimit,tor_label(Xs))
          ,dbs(YsLimit,tor_label(Ys)))).
```

Note that the global variables necessary for the two depth bounds exist simultaneously, but, because we use different ones, they do not interfere.

Finally, one handler can relinquish control to another one. For instance, inspired by ECLiPSe's search/6 predicate, we provide a variant dbs/3 of depth-bounded search that does not fail when the bound is reached, but switches to another strategy. While ECLiPSe only offers two alternatives for the other strategy, Tor allows any handler. For instance, once the depth-limit is reached, we visit only a fixed number of nodes with:

```
?- ...,
   search(dbs(DepthLimit
             ,nbs(NodeLimit,tor_label(Vars)))).
```

6. Plain Prolog Example

While the application of Tor to CLP problems is obvious, we wish to emphasize that Tor is not limited to CLP.

For that reason we illustrate the use of Tor on the well-known problem of the wolf, the goat and the cabbage. The following code, adapted from Sterling and Shapiro [16], implements this decision problem in plain Prolog (without constraints). Naive depth-first execution of this code loops infinitely.

```
wgc :-
  initial_state(State),
  wgc(State).

wgc(State) :-
  final_state(State), !.
wgc(State) :-
  move(State,Move),
  update(State,Move,State1),
  legal(State1),
  wgc(State1).

initial_state(wgc(left, [wolf, goat, cabbage], [])).

final_state(wgc(right, [], [wolf, goat, cabbage])).

move(wgc(Bank, Left, Right),Move) :-
  ( Bank == left,
    tor_member(Move, Left)
  tor
    Bank == right,
    tor_member(Move, Right )
  tor
    Move = alone
  ).

:- tor tor_member/2.
tor_member(X,[X|_]).
tor_member(X,[_|Xs]) :- tor_member(X,Xs).

update(wgc(B,L,R), Cargo, wgc(B1, L1, R1)) :-
  update_boat(B, B1),
  update_banks(Cargo, B, L, R, L1, R1).

update_boat(left, right).
update_boat(right, left).

update_banks(alone, _B, L, R, L, R) :- !.
update_banks(Cargo, left, L, R, L1, R1) :- !,
  select(Cargo, L, L1),
  insert(Cargo, R, R1).
update_banks(Cargo, right, L, R, L1, R1) :-
  select(Cargo, R, R1),
  insert(Cargo, L, L1).

insert(X,[Y|Ys], [X,Y|Ys]) :-
  precedes(X,Y), !.
insert(X, [Y|Ys], [Y|Zs]) :-
  precedes(Y,X), !,
  insert(X,Ys,Zs).
insert(X, [], [X]).

precedes(wolf, _X).
precedes(_X, cabbage).

legal(wgc(left, _L, R)) :- \+ illegal(R).
legal(wgc(right, L, _R)) :- \+ illegal(L).

illegal(Bank) :- memberchk(wolf, Bank),
                 memberchk(goat, Bank).
illegal(Bank) :- memberchk(goat, Bank),
                 memberchk(cabbage, Bank).
```

The nondeterministic enumeration in this code is situated in the move/2 and tor_member/2 predicates. In order to use Tor, we have replaced ordinary Prolog disjunction with tor/2.

To avoid the non-termination, we can apply a depth-bound and discover in finite time that the problem has a solution.

```
?- search(dbs(17,wgc)).
true.
```

7. Evaluation

To study Tor's overhead, we have performed a number of benchmarks on a MacBook Pro with a 2.4 GHz CPU and 4 GB RAM. We compare two Prolog systems with different performance characteristics. On the one hand we consider SWI-Prolog 5.11.7 in Mac OS X 10.6.7, a feature-rich, but relatively slow Prolog system with a CLP(FD) solver written in Prolog. On the other hand, we consider B-Prolog 7.5#3, one of the fastest Prolog systems with a highly optimized CLP(FD) implementation.

7.1 Pure Search

Figure 4 considers the extreme situation where the search is pure enumeration of *unconstrained* constraint variables: length(N,Vars), Vars ins 1..D. Hence, no constraint propagators are activated due to choices. Values are simply enumerated.

The first column denotes the problem size, expressed in the number of variables N and their domain size D. The other three pairs of columns denote different implementations of labeling: 1) label/1 as listed in this paper, 2) label/1 from SWI-Prolog's clpfd library and the corresponding labeling/1 provided by B-Prolog, and 3) search/6 ported from ECLiPSe to SWI-Prolog and B-Prolog with minimal changes. For each of these, we show the absolute runtime of the standard/manual version (man) and the relative runtime of the Tor version (tor).

In both SWI-Prolog and B-Prolog the impact of Tor is pretty consistent across the problem sizes, but depends on the labeling implementation. In SWI-Prolog, the overhead is most prominent (140-180 %) in our barebones label/1, while it is less so (50-60 %) in clpfd's label/1. The latter delegates to labeling/2, which in-

	our `label/1`		`clpfd`'s `label/1` B-Prolog's `labeling/1`		`search/6`	
	man	Tor	man	Tor	man	Tor
SWI-Prolog						
`N=6,D= 8`	1.80 s	240 %	2.08 s	151 %	2.55 s	132 %
`N=6,D= 9`	3.63 s	249 %	4.20 s	153 %	5.09 s	135 %
`N=6,D=10`	6.82 s	269 %	7.87 s	155 %	9.53 s	137 %
`N=7,D= 8`	14.44 s	244 %	16.63 s	153 %	20.40 s	134 %
`N=7,D= 9`	32.80 s	269 %	37.80 s	155 %	46.04 s	136 %
`N=7,D=10`	68.27 s	278 %	78.63 s	157 %	94.30 s	139 %
B-Prolog						
`N=6,D= 8`	0.49 s	156 %	0.09 s	276 %	0.12 s	223 %
`N=6,D= 9`	0.99 s	157 %	0.18 s	283 %	0.23 s	221 %
`N=6,D=10`	1.87 s	160 %	0.32 s	291 %	0.44 s	219 %
`N=7,D= 8`	4.56 s	144 %	0.71 s	306 %	0.94 s	220 %
`N=7,D= 9`	8.90 s	163 %	1.59 s	301 %	2.06 s	225 %
`N=7,D=10`	18.64 s	163 %	3.25 s	332 %	4.37 s	220 %

Figure 4. Labeling benchmarks without propagation

volves more generic option processing. Finally, in `search/6` Tor compensates its overhead further (to 30-40 %) by not collecting search statistics when these are not demanded. In the original version, these statistics are collected regardless of demand.

In B-Prolog, the performance characteristics of the labeling predicates are markedly different. Firstly, the cost of the inequality `(#\=)/2` in our `label/1` is relatively high, which keeps the overhead of Tor low (60%). In contrast, the two other labeling predicates rely on B-Prolog's `domain_inst_next/3` for enumeration, which compiles down to a single machine instruction. As a result the overhead of Tor is much higher, more so in the tight `labeling/1` (170%-230%) than the more bloated `search/6` (120%).

In summary, in these propagation-free benchmarks, the overhead of Tor goes up to about a factor three for tight labeling loops, but is lower for option-rich labeling predicates. Moreover, Tor is better behaved in SWI-Prolog than in B-Prolog. All in all, we find that this is a very reasonable price to pay for the extra flexibility that Tor provides. Still, invoking Tor's specializer (see the next section) is warranted to get rid of all overhead.

7.2 Search vs. Propagation

While the overhead of Tor is bounded in the previous benchmarks, the performance-wary user may not be willing to accept the overhead. However,the previous benchmarks are not representative of realistic CLP problems, that spend a lot of time on constraint propagation in every node of the search tree. All this extra work easily dwarfs the overhead of Tor. Figure 5 illustrates this observation on a number of typical CLP benchmarks.

For added realism, the benchmarks use the first-fail variable selection strategy, with hand-written labeling code `ff_label/1`, the two library predicates `labeling/2` (SWI-Prolog) and `labeling_ff/1` (B-Prolog), and the ported `search/6`. Because B-Prolog's CLP(FD) solver is orders of magnitude faster than SWI-Prolog's, it makes little sense to use exactly the same benchmarks for the two platforms. Instead, we resorted to different problem sizes or different benchmarks altogether.

	plain	lds	dibs-1	dibs-2	credit/bbs
`N= 95`	2.11 s	0.66 s	0.45 s	**0.28 s**	0.33 s
`N= 96`	**0.65 s**	4.98 s	4.89 s	1.13 s	1,04 s
`N= 97`	T/O	3.68 s	**3.56 s**	22.66 s	4,08 s
`N= 98`	T/O	15.67 s	† 5.71 s	10.16 s	**2.50 s**
`N= 99`	T/O	2.42 s	**2.22 s**	9.85 s	2.57 s

† no solution

Figure 6. N-Queens benchmarks with various search methods

In the case of SWI-Prolog, we see that Tor introduces no (significant) overhead; its runtime is marginal compared to that of constraint propagation. In the case of B-Prolog, the overhead of Tor is more noticeable, in the order of 10% for most benchmarks. Only in the case of the knapsack problem does it go up to 75% for the tightest labeling loop.

In summary, we see little reason not to use Tor for most CLP problems. Especially in SWI-Prolog there is no runtime price to pay. In the setting of B-Prolog, an extra 10% runtime is a cheap price for the extra flexibility that Tor provides. Moreover, in the next section we will see how we can eliminate Tor's overhead to the extent that we don't pay for it if we don't use its extra flexibility.

7.3 Search Methods

Finally, Figure 6 illustrates again why we want to use different search methods: they can have a significant impact on runtime. The figure shows the runtime of the **n_queens** benchmark in SWI-Prolog for 5 different problem sizes and 5 different search methods: (plain) plain depth-first search, (lds) limited discrepancy search, (dibs-1/-2) discrepancy bounds of 1 and 2, and (credit/bbs) credit-based search with 10,000 credits that switches to a bounded backtracking (1 backtrack) search when the credits are exhausted.

	our `ff_label/1`		`labeling/2`		`search/6`	
	man	Tor	man	Tor	man	Tor
SWI-Prolog						
`allinterval`	4.03 s	101 %	4.02 s	101 %	4.01 s	101 %
`golf`	3.93 s	99 %	3.92 s	100 %	3.96 s	99 %
`mhex`	18.59 s	102 %	18.61 s	101 %	18.46 s	101 %
`n_queens`	2.03 s	103 %	2.05 s	102 %	2.09 s	102 %
`sudoku`	2.14 s	101 %	2.15 s	101 %	3.40 s	100 %
B-Prolog						
`allinterval`	1.14 s	100 %	0.81 s	112 %	0.89 s	109 %
`knapsack`	3.94 s	125 %	2.11 s	175 %	2.17 s	172 %
`knight`	0.67 s	101 %	0.71 s	100 %	0.91 s	100 %
`mhex`	0.23 s	106 %	0.19 s	107 %	0.23 s	104 %
`n_queens`	1.01 s	107 %	0.89 s	107 %	1.03 s	106 %

Figure 5. Labeling benchmarks with propagation

8. Automatic Specialization

Tor encourages writing fairly abstract and generic code. This style clearly incurs some overhead (notably due to meta-calling) compared to specialized search code. Fortunately, in the case of CLP applications, this overhead is very modest compared to the bottleneck of constraint propagation. However, in the case of applications without constraint propagation, we do observe an overhead that is not insignificant. In order to mitigate that overhead, we exploit Prolog's homoiconic nature to provide a simple but effective automatic specializer.

Even though there is a large body of work on automatic program specialization for Prolog, notably involving partial evaluation, we decided to write our own program specializer. Its main tasks are 1) to perform *constant propagation* on the global variables `left` and `right`, 2) to replace instantiated meta-calls by direct calls and 3) to inline the handler code into the main search loop. For control we follow a light-weight approach based on declarations of what predicates to inline and specialize.

Example 1 Our specializer yields `label/1` for the generic composition `search(tor_label(Vars))`. Similarly, we recover SWI-Prolog's `labeling/2` by specializing its Tor variant. Hence, we do not pay if we do not use search methods.

Example 2 The specialized form of the goal `search(dbs(N, tor_label(Vars)))` is `new_bvar(N,DVar), label21(Vars, DVar)`, with:

```
label21([], _).
label21([Var|Vars], DVar) :-
  ( var(Var) ->
      fd_inf(Var, Val),
      ( b_get(DVar, Depth),
        Depth>0,
        NDepth is Depth+ -1,
        b_put(DVar, NDepth),
        Var#=Val,
        label21(Vars, DVar)
      ;
        b_get(DVar, G),
        G>0,
        NDepth is G+ -1,
        b_put(DVar, NDepth),
        Var#\=Val,
        label21([Var|Vars], DVar)
      )
  ;
      label21(Vars, DVar)
  ).
```

This code is slightly less efficient than that of `label/2`. Firstly, the overhead of mutable variables is not entirely eliminated here, as `DVar` is still present. Secondly, the two branches have some code in common that could be shared. However, there are no more meta-calls and all code is inlined in the recursive loop of `label21/2`.

In future work, we intend to get rid of the remaining inefficiencies by implementing additional transformations, including Peter Schachte's approach [12] for eliminating mutable variables adapted to our setting.

9. Related Work

We have already covered the most closely related work, existing approaches to search heuristics in Prolog, in Section 2.2. Here we cover other important related topics.

Earlier Work Tor is related to earlier work on *Monadic Constraint Programming* (MCP) [14] in the context of Haskell, and *Search Combinators* [15] in the context of C++ and the Gecode library[5]. In contrast to those works, Tor is tailored towards Prolog's built-in depth-first search and, as a consequence, consists of a much simpler and more elegant design.

Comet The imperative Comet language [19] features fully programmable search by means of *search controllers* [20]. There are too main differences between Tor and Comet's search controllers. Firstly, search controllers exchange simplicity for flexibility, providing more hooks and first-class continuations to manipulate the search. Secondly, search controllers are not intended to be used together, in contrast to Tor's handlers that are explicitly designed to be composed.

Aspect-Oriented Programming The Tor approach is closely related to Aspect-Oriented Programming (AOP) [8, 10]. AOP provides a generic approach for modularly *crosscutting* existing code

[5] `http://www.gecode.org`

with new code, so-called *advice*. This advice is injected in arbitrary *join points* (i.e., program points) based on a *pointcut* predicate.

Obviously Tor is more limited in scope, as only `tor/2` disjunctions are crosscut and only at the positions of the two hooks. However, we believe that these "limitations" are actually Tor's strength: its simplicity makes it easy to express all common search methods and its discipline favors compositionality.

10. Design Discussion

In this section we briefly discuss a number of design decisions of Tor as well as possible future extensions.

Increased Expressivity Simplicity has been a guiding principles in the design of Tor. In order to minimize the threshold for users, we keep the effort and complexity of defining and using search methods low. We pay for this simplicity with somewhat restricted expressivity. An example of a search method that cannot be expressed with Tor is swapping the order of branches in a disjunction. In order to overcome this limitation we would have to add extra complexity to the `tor/2` built-in in the form of an additional hook. Nevertheless, Tor is remarkably expressive as it is, covering all of the commonly found search methods in CLP(FD) libraries.

Multiway Disjunctions Tor currently only supports binary disjunctions; multiway disjunctions have to be decomposed into binary ones. For some applications, this decomposition can be somewhat unnatural. For instance, when enumerating all the values `V` of a constraint variable `X`, one might expect that all alternative assignments `X #= V` sit at the same level in the search tree. This is of course generally not the case in a binary decomposition. For that reason we are considering backward compatible ways to generalize the handler approach.

Declarative State Management We have already mitigated the dangers of mutable variables with reference cells. Nevertheless handler programming requires a rather imperative programming style. We are considering more declarative interfaces, inspired by definite clause grammars and the state monad [21], that better screen the handler programmer from the mutation side effect. The idea is to add behind the scenes an impure wrapper around pure handler code. There are two complicating factors in the matter: Tor's higher-order programming style and non-backtrackable state. The higher-order style complicates the boundaries of the handler code and thus increases the complexity of a pure interface to it. Non-backtrackable state updates are often followed immediately by failure. There is no idiomatic declarative alternative for this technique. However, we can turn to pure deterministic encodings of failure with non-backtrackable state, like Haskell's `ListT (State s)` monad [7] and use Filinski's reification/reflection technique [4] to translate to and from Prolog's native effects.

11. Conclusion

We have presented Tor, a light-weight library-based approach for modifying Prolog's depth-first search with reusable and compositional search methods.

On a more drastic account, we will investigate ways to replace the underlying depth-first queuing strategy. The stack freezing functionality of tabling systems like XSB [17] and YAP [11] provides interesting perspectives for this purpose.

Acknowledgements We are grateful to Rémy Haemmerlé and Jose Morales Caballero for our discussions on Tor, and to Neng-Fa Zhou for revealing B-Prolog's `labeling/1` code to us.

References

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Time stamps techniques for the trailed data in constraint logic programming systems. In Serge Bourgault and Mehmet Dincbas, editors, *SPLT'90, 8ème Séminaire Programmation en Logique, 16-18 mai 1990, Trégastel, France*, pages 487–510, 1990.

[2] Mats Carlsson and Per Mildner. SICStus Prolog - The first 25 years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012.

[3] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):253–282, 2012.

[4] Andrzej Filinski. Monads in action. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 483–494. ACM, 2010.

[5] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 15th International Joint Conferences on Artificial Intelligence (IJCAI 1995)*, pages 607–613, 1995.

[6] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and German Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[7] Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, Department of Computer Science, New Haven, Connecticut, December 1993.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Christina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, pages 220–242, 1997.

[9] Robert Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

[10] Wolfgang Lohmann, Günter Riedewald, and Guido Wachsmuth. Aspect-Orientation in Prolog. In *Proceedings of the 16th International Symposium on Logic-based Program Synthesis and Transformation*, 2006.

[11] Vitor Santos Costa, Ricardo Rocha, and Luis Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.

[12] Peter Schachte. Global variables in logic programming. In *Proceedings of the International Conference on Logic Programming (ICLP 1997)*, pages 3–17, 1997.

[13] Joachim Schimpf and Kish Shen. ECLiPSe From LP to CLP. *Theory and Practice of Logic Programming*, 12(1-2):127–156, 2012.

[14] Tom Schrijvers, Peter J. Stuckey, and Philip Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663–697, 2009.

[15] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter Stuckey. Search Combinators. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011)*, volume 6876 of *Lecture Notes in Computer Science*, pages 774–788. Springer, 2011.

[16] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, 2. edition, 1994.

[17] Terrance Swift and David S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[18] Markus Triska. The finite domain constraint solver of SWI-Prolog. In *Proceedings of the 11th International Symposium on Functional and Logic Programming (FLOPS 2012)*, pages 307–316, 2012.

[19] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. MIT Press, 2005.

[20] Pascal Van Hentenryck and Laurent Michel. Nondeterministic control for hybrid search. *Constraints*, 11(4):353–373, 2006.

[21] Philip Wadler. The essence of functional programming. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1992)*, pages 1–14. ACM Press, 1992.

[22] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[23] Neng-Fa Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, 2012.